iota - эффективный путь для объявления констант, в частности, для описания перечислений.

Мы постоянно говорим о том, что программный код должен быть содержательным, что он должен отражать наши соображения по организации обработки данных, что идентификаторы следует выбирать так, чтобы они как-то отражали соответствующие им понятия. Часто очень важно заботиться об именах понятий в программном коде.
Бывают моменты, когда конкретные значения каких-то величин не имеют никакой внутренне значимой ценности, что нам неважны конкретные значения, нам важно только отличать одну величину от другой.

Например, если мы храним продукты в таблице базы данных, мы, вероятно, не хотим хранить их категорию в виде строки. Нам все равно, как называются категории, и, кроме того, маркетинг постоянно меняет названия.

Ниже приводятся несколько примеров, которые показывают этакое абстрактное применение iota и некоторые её особенности.

iota - простейший пример, упрощение записи - распространение iota по умолчанию, перезагрузка iota, в простейшем случае тип получаемых констант = int

iota последовательно принимает значения 0, 1, 2, …, увеличиваясь на 1 в каждой следующей строке объявления констант.

Каждый раз, когда в программном коде появляется ключевое слово const, iota перезагружается - получает значение 0.

Запись может быть упрощена - явно достаточно записать присваивание iota константе только один раз.

Правило хорошего тона - заводить функцию, которая по числовому значению константы выдаёт текст - описание, название константы. Получая int, возвращает string.

Эти свойства iota показывает пример `iota01.go` :

```
package main

import "fmt"

func main() {
    // Basic sample
    const (
        Start = iota
        Run = iota
        End = iota
    )
    fmt.Printf("Start: %d\nRun: %d\nEnd: %d\n", Start, Run, End)
    fmt.Println()
/*
Start: 0
Run: 1
End: 2
*/

    // Simplified notation.
    //     Every time keyword const appears in the code, iota's value is reseting
    const (
        Small = iota
        Medium
        Large
    )
    fmt.Printf("Small: %d %T\nMedium: %d %T\nLarge: %d %T\n", Small, Small, Medium,
Medium, Large, Large)
    fmt.Println()
/*
Small: 0 int
Medium: 1 int
Large: 2 int
*/

const (
    Black = iota
    Blue
    Green
```

```
    Cyan
    Red
    Magenta
    Brown
    Gray
)
    fmt.Printf("Colors:\n Black = %d %T\n Blue = %d %T\n Green = %d %T\n Cyan = %d %
T\n" +
               " Red = %d %T\n Magenta = %d %T\n Brown = %d %T\n Gray = %d %T\n",
                Black, Black, Blue, Blue, Green, Green, Cyan, Cyan,
                Red, Red, Magenta, Magenta, Brown, Brown, Gray, Gray)
    fmt.Println()
/*
Colors:
 Black = 0 int
 Blue = 1 int
 Green = 2 int
 Cyan = 3 int
 Red = 4 int
 Magenta = 5 int
 Brown = 6 int
 Gray = 7 int
*/

// Best practice: complete enumerated type (enum) with strings.
// See function ColorName definition below.
    for c:= 0; c < 8; c++ {
        fmt.Printf("%d %T %s\n", c, c, ColorName(c) )
    }
    fmt.Println()
/*
0 int Black
1 int Blue
2 int Green
3 int Cyan
4 int Red
5 int Magenta
6 int Brown
7 int Gray
*/
```

```
}

func ColorName(c int) string {
    names:= [...]string{"Black", "Blue", "Green", "Cyan", "Red", "Magenta", "Brown",
 "Gray"}
    return names[c]
}
```

iota и формулы, распространение формул и типов; пропуск значений iota

- При объявлении констант допускается выполнять какие-то операции с iota. При этом действие формулы распространяется на все следующие строки вплоть до объявления константы с другой формулой. В частности, присвоив константе в первой строке значение выражения iota+1, можно получить набор констант, пронумерованных с 1, а не с нуля (1-based enumeration).
- iota - бестиповая константа, может быть присвоена любому числовому типу. Тип получающейся константы можно задать явно. При этом указание типа константы распространяется, подобно формуле, на все следующие строки вплоть до объявления константы с другой формулой. Именно формулой, указать тип константы без формулы нельзя.
- если в какой-то строке iota не используется, например, константе присваивается некое константное значение или выражение, используется пустой идентификатор `_` (blank identifier), то значение iota, хоть и игнорируется, но всё равно увеличивается на 1 в этой строке.

Эти ситуации иллюстрируются примером `iota02.go`

```
package main

import "fmt"

// It’s possible to make operation when using iotas
// Operation's expansion
const (
    n         = iota
    k
    l int8      = iota + 10
    m
    x float32 = iota * 0.3
```

```
    y
    z
    u         = iota * 3.1
    v
    w
)
/*
n: 0 — int
k: 1 — int
l: 12 — int8
m: 13 — int8
x: 1.2 — float32
y: 1.5 — float32
z: 1.8 — float32
u: 21.7 — float64
v: 24.8 — float64
w: 27.9 — float64
*/

// Start from one
const  (
    level1 = iota + 1
    level2
    level3
)
/*
level1: 1 — int
level2: 2 — int
level3: 3 — int
*/

// Skip iota's value
const (
    a           = iota * 3 - 1
    b

    _
    c uint8   = 8

    _
    d
    e          = iota
)
```

```
/*
a: -1 — int    // iota = 0: 0*3 - 1
b: 2 — int     // iota = 1: 1*3 - 1
               // skip iota, iota = 2
c: 8 — uint8   // iota = 3
               // skip iota, iota = 4
d: 8 — uint8   // iota = 5
e: 6 — int     // iota = 6
*/

func main() {
   fmt.Printf("n: %v — %T\n", n, n)
   fmt.Printf("k: %v — %T\n", k, k)
   fmt.Printf("l: %v — %T\n", l, l)
   fmt.Printf("m: %v — %T\n", m, m)
   fmt.Printf("x: %v — %T\n", x, x)
   fmt.Printf("y: %v — %T\n", y, y)
   fmt.Printf("z: %v — %T\n", z, z)
   fmt.Printf("u: %v — %T\n", u, u)
   fmt.Printf("v: %v — %T\n", v, v)
   fmt.Printf("w: %v — %T\n", w, w)
   fmt.Printf("level1: %v — %T\n", level1, level1)
   fmt.Printf("level2: %v — %T\n", level2, level2)
   fmt.Printf("level3: %v — %T\n", level3, level3)
   fmt.Printf("a: %v — %T\n", a, a)
   fmt.Printf("b: %v — %T\n", b, b)
   fmt.Printf("c: %v — %T\n", c, c)
   fmt.Printf("d: %v — %T\n", d, d)
   fmt.Printf("e: %v — %T\n", e, e)
}
```

действие iota начинается с самого начала раздела описания констант; iota изменяется только при переходе к следующей строке

- да, даже если iota впервые в разделе встретилась не в первой строке, а все строки, находящиеся перед первым появлением iota в одном разделе const, присваивают контантам какие-то выражения, не содержащие iota, то всё действие iota распространяется вверх до перпвого появления iota; всё равно iota в верхней строке имеет значение 0, которое в каждой следующей строке увеличивается на 1.

- выражения, содержащиеся в одном списке выражений, используют одно и то же значение iota. iota изменяет значение только при переходе к следующему выражению или списку выражений.
- список выражений распространяется точно также, как и отдельная формула.

Эти ситуации уже немножко экзотичны, может быть не стОит про них говорить, а просто дать детям задание поэкспериментировать над iota, измываясь над ней, как только возможно, но для затравки этого задания пример `iota03.go`, иллюстрирующий такие ситуации, покатит. Это, кстати, очень уместно делать в Go Playground'е.

```
package main

import "fmt"


const  (
    stub = -10000
    level1 = iota
    level2
    level3
)
/*
level1: 1 – int
level2: 2 – int
level3: 3 – int
*/

const  (
    alpha1, beta1 = iota, iota*3
    alpha2, beta2
    alpha3, beta3
)
/*
alpha1: 0 – int;  beta1: 0 – int
alpha2: 1 – int;  beta2: 3 – int
alpha3: 2 – int;  beta3: 6 – int
*/

func main() {
   fmt.Printf("level1: %v – %T\n", level1, level1)
   fmt.Printf("level2: %v – %T\n", level2, level2)
```

```
    fmt.Printf("level3: %v – %T\n", level3, level3)
    fmt.Printf("alpha1: %v – %T;  ", alpha1, alpha1)
    fmt.Printf("beta1: %v – %T\n", beta1, beta1)
    fmt.Printf("alpha2: %v – %T;  ", alpha2, alpha2)
    fmt.Printf("beta2: %v – %T\n", beta2, beta2)
    fmt.Printf("alpha3: %v – %T;  ", alpha3, alpha3)
    fmt.Printf("beta3: %v – %T\n", beta3, beta3)
}
```